Summary of the DH IO Modules Review

Marc Paterno

November 2, 2001

Abstract

This is a brief summary of the agreements reached during the review of the DH IO modules, from meetings on 17 and 19 September, 2001. Included in the appendix are comments on the a subset of the classes in the *DHMods* package.

1 Introduction

This review was held to help resolve some technical issues regarding the design of the framework modules which use the Data Handling system to perform event-related input and output. The participants in the review were: Rob Kennedy, Fedor Ratnikov, Pasha Murat, Rob Harris, Liz Sexton-Kennedy, and myself. In the interest of brevity, I will not try to summarize the positions of all the interested parties *before* the review meetings. I summarize here only the agreements reached during the meetings, and add a few comments of my own concerning how some of the unresolved issues may be resolved.

I was also asked to perform a brief code review of a subset of the classes in the *DHMods* package. My comments can be found in the appendix. These comments are my opinions only, and do not (necessarily) reflect the opinions of the rest of the review group.

2 Meeting Summary

The main issue to be resolved revolved around the choice of which software package (*DHMods*, *Edm*, or *Framework*) should be responsible for various functionalities. Fedor provided a list of functionalities, and after some discussion we agreed on the following package assignments. In several cases, the required functionality needs to be coordinated between different packages; these cases are noted, and the responsible parties are listed.

1. Selection of data by any combination of dataset or fileset or file names. This is a *DHMods* responsibility.

2. "Include" and "exclude" support for datasets, filesets, and files. This is a *DHMods* responsibility.

3. Placement of restrictions on run numbers and runsection numbers upon input. This is a *DHMods* responsibility.

4. Provision of access to data residing in the DH system and to local "private" files. This is a *DHMods* responsibility. At the appropriate time, it may be possible to make the *FileInputModule* obsolete, in order to reduce the maintenance burden associated with the IO modules. This will be sensible to consider after the unification of *CDFRootFileStream* and *SeqRootDiskFile* classes.

5. Communication with the Data File Catalog to obtain a full list of requested data. This is a *DHMods* responsibility.

6. Communication with the Disk Inventory Manager to deliver requested data in the most effective order. This is a *DHMods* responsibility.

7. Processing of events in the "natural" order, and building a catalog of events in the file. This is an *Edm* responsibility.

8. Navigation within a single file; skipping events both forwards and backwards, direct access by run and event number, inserting Begin of Run records as necessary. This is an *Edm* responsibility.

9. Reading and writing events using ROOT buffers, without expanding objects. This is an *Edm* responsibility.

10. Filtering of input events, by run number and event number. This is a *Framework* responsibility.

11. Specification of output by file name or dataset name. This is a *DHMods* responsibility.

12. Assignment of data file names according to the CDF convention. This is a *DHMods* responsibility.

13. Collection of statistics (number of bytes written, *etc.*) for the output file. This is an *Edm* responsibility.

14. Collection of output files in a given directory. This responsibility needs to be clarified; the proper assignment was not clear. One suggestion is that the collection of output files in a given directory should be managed by *DHMods*, which perhaps directs the actions of *Edm* classes to perform the specific tasks.

15. Entering of FILE record into the Data File Catalog. This is a *DHMods* responsibility.

16. Splitting of output data into files of a given size. This is an *Edm* responsibility, but the *Edm* is not the correct place for the information defining CDF policies to reside. The *Edm* needs to be given the correct information to control this functionality. The details need to be agreed upon by Rob and Fedor.

17. Keeping runsections contiguous in a file. This is an *Edm* responsibility, but needs information from higher-level packages to do the necessary work. The details need to be agreed upon by Rob and Fedor.

18. Intermediate saving of file status, to minimize reprocessing in case of the crashing of a job. This is an *Edm* responsibility.

19. Creation of new ERS (empty runsection) when an EmtpyRunsection condition is detected. The ERS itself belongs to the *Edm*, but the triggering of its creation is a *DHMods* responsibility, since it has the DFC access required for this function.

20. Creation of multiple data branches, synchronized with the primary data branch. This is an *Edm* responsibility.

21. Keeping the Data File Catalog consistent in case of job crashes. This is a *DHMods* responsibility. In order to obtain full automation, some of the required functionality may require additions to *Framework*. Fedor and Liz should agree upon the details.

An important general rule that guided these decisions is that handling of *files* is the responsibility of *DHMods*, but the contents of those files is the responsibility of the *Edm*.

It was agreed that responsibilities in the list above which properly belong to the *Edm* will be integrated into the *Edm*, and that the *DHMods* classes which contain this functionality will become obsolete once the *Edm* implementations are ready. The detailed coordination of the move is left to Rob and Fedor.

A Some Comments on Classes in *DHMods*

I was asked to perform a brief code review for the classes in *DHMods*. This was limited to only those classes which will *not* become obsolete due to the realignment of responsibilities. I have concentrated on issues of maintainability, and thus on behavior that may be surprising to users or future maintainers of these classes.

I have significant reservations about this package. I find problems in both high-level design and low-level details. For example, in the area of high-level design, the classes *DHInputModule* and *DHOutputModule*

both significantly violate the idea of a modular framework; they communicate directly with each other, which to a large extent prevents their interchangeable use with other input and output modules. This flaw is probably the single most significant in the design; it *must* be changed. If the solution requires addition to the framework, that development must be coordinated with the framework, and not implemented in the *DHMods* package.

A recurring design defect is the introduction of needless inheritance hierarchies. An inheritance hierarchy is a needless complication when it contains only one derived class, and when there is no clear need for a second derived class. In this situation, the extra complexity is a detriment to maintainability, and is sometimes a detriment to efficiency. It also makes testing more complex, since the general interface should be tested separately from the single implementation.

As an example of problems in the low-level implementation details, I have inspected the class *DHInputModule* in a fair level of detail, and in the section below present many comments on it.

For want of time, I have treated the other classes in much less detail. The comments I make on the design and implementation of *DHInputModule* are in many cases apply to these other classes.

Finally, there is insufficient documentation of most classes in the code. The CDF coding guidelines sensibly require a short description of the *purpose* of each class in the header which declares the class. The common lack of such descriptions made the review of this package significantly more time-consuming than it would otherwise have been. This will also be a sever detriment to the maintainability of this package.

A.1 *DHInputModule*

DHInputModule.hh The function

```
virtual std::string DHInputModule::description() const;
```

is a dangerous override to the base class function

```
char* APPExecutable::description() const;
```

This function has several problems. Note that the function in *DHInputModule* has the same name and argument types, so it is an *override*, not an *overload*. The base class function is not virtual, so polymorphic behavior will not occur when the function is invoked on a base class pointer. When the function is invoked on an object which has a static type of *DHInputModule*, polymorphic behavior will occur. The return type of the derived class's function is also different from that of

the base class's function, and the types are not implicitly convertible, nor are they related by derivation. These are all dangerous features. The function in *DHInputModule* should be removed.[1]

The function

```
virtual inline std::string eventBranchName ()
   { return "Sequential"; }
```

has several problems.

First, since the function is implicitly inlined (it appears in the body of the class declaration), the keyword *inline* is not needed. Second, it is only under exceptional circumstances that a compiler is able to inline a call to a function that has been declared virtual. The usual result is to *not* inline the function call, but to still suffer some code bloat for the function which will appear in every compilation unit that includes the header in which the function is declared. Generally, a function should not be declared both *virtual* and *inline*.

In this case, it seems that polymorphic use of `eventBranchName` is unlikely; if this is correct, then the function should not be declared virtual. If in fact polymorphic use of this function is likely, then the function should not be inlined, and should be implemented in the the source (not header) file for this class.

Functions that do not modify the object on which they are invoked, and which do not return pointers or references to members through which the object may be modified, should be declared *const*. For example, the function

```
const AbsEvent* getConstRecord ();
```

does not modify the state of the *DHInputModule*, nor does it provide non-const access to internal data; it should be declared const.

Because of the resolution of responsibilities described in §2, the following functions should probably be removed:

```
virtual bool jumpToEvent (int fRun, int fEvent);
virtual bool jump (int fNRecords);
virtual const EventInfoBase*  nextRecord ();
```

It is possible that the same resolution of responsibilities makes some of the data members of *DHInputModule* superfluous; for example, the need for `TBuffer* _mCompactBuffer` seems to have been removed. All the data members should be reconsidered.

[1] It would also be best to modify the function in *APPExecutable* to return a pointer to *const*, or, if the resulting changes in client code are not too great, a *std::string* or const reference to a *std::string*.

DHInputModule.cc The use of the non-standard *long long* (which is used through the typedef *int8*) would be better replaced with a class with the appropriate interface. This would avoid code such as:

```
int8 runSection (unsigned long fRun,
                 unsigned short fSection) {
    int8 result = fRun & 0xffffff;
    result = (result << 16) | (fSection & 0xffff);
    return result;
}
```

This is, strictly speaking, not a *DHInputModule* issue; *int8* would be better implemented as a class in the same file that now contains the typedef. This class should provide an interface that makes the explicit bit-twiddling in this function unnecessary.

The handling of the data members `_mRootFileStream` and `_theFile` (inherited from the base class *APPReaderInputModule*) is extremely confusing and dangerous. From the base class we inherit a pointer to an *AbstractFile*; the pointed-to object is owned by the *APPReaderInputModule*, and is deleted in the destructor. Setting this pointer to be the address of another data member of *DHInputModule* will result in double destruction of the pointed-to object, with undefined (and probably disastrous) behavior. This is currently avoided by having the base class's data member set to zero during destruction, but this is not good lifetime management practice, and seems very likely to lead to some future bugs (if, for example, the base class is modified).

If the requirement is to have access to the interface of the *CDFRootFileStream* class, and to avoid typing dynamic_cast in many functions, then a private member function that wraps the dynamic_cast might be appropriate. It is my understanding that the *CDFRootFileStream* class will be replaced with a newer class from the EDM; at the soonest possible time this should be done.

As a minor point for efficiency and safety, it would be better for the many data members of *DHInputModule* to be set in the colon-initialization list for the class, rather than having them re-assigned in the body of the constructor.

The destructor of *DHInputModule* is also much more complicated than necessary, which makes it harder to maintain. Since it is safe to delete a null pointer, there is no need to test the pointer before calling delete on it. And since the object containing the pointer is about to disappear, there is no need to zero the pointer after deletion.

Thus the repeating blocks similar to:

```
if (_mCompactBuffer) {
    delete _mCompactBuffer;
    _mCompactBuffer = 0;
```

```
}
```

can be replaced by the single line

```
delete _mCompactBuffer;
```

This complication is repeated in other classes in *DHMods*; is should be removed wherever it is found.

Much of the work done in `beginJob` is likely to be changed when the management of files and buffers is given over to the appropriate EDM classes. My main concern with this function is the calling of the base class function `beginJob`. It seems likely that the base class functions were written to override, rather than to cascade. If derived classes really need to use the functionality in the base class, and to add a bit of their own, then I would recommend use of the *Template Method* pattern[2]. The same pattern may also be appropriate for `endJob` and `abortJob`.

Also in `abortJob`, writing to `std::cout` and `std::cerr` should be replaced by use of the *ErrorLogger*. This is repeated in many functions; the module should be searched for all such instances (except where the output is to `std::cout` for menus, or other interactive use where output to the terminal is really wanted).

The function `openNextFile` is very difficult to follow, and could use refactoring for the sake of maintainability. For example, I believe that if the test

```
AbsInterp::theInterpreter()->
  fileOperation(_theFilename.c_str(),"exists" ))
```

fails, `openNextFile` will return `AppResult::EOFR`, indicating a normal end-of-file has been reached. It seems likely that this is not correct behavior, but the structure of the code is such that it is not immediately clear that this is the result. It also seems that there should be some more sensible way to test for the existence of a file than invoking an interpreter with the appropriate “magic strings”, but this may be outside the control of *DHMods*.

The function `nextEvent` is still more complex than `openNextFile`; it extends for about 160 lines, as a series of nested `if` tests. It is in clear need of refactoring into manageable pieces. A block of code this long and complex is almost certainly too complicated to test.

It seems very odd that *DHInputModule* would need a function like `getOutputModule`, `getDHOutputModule`, or `sendEmptyRunSections-ToOutput`, or any of the several other output functions . The job of *DH-InputModule* is to perform input – why is it writing to the output? Unless my understanding of this basic feature is incorrect, it may be that

[2] *Template Method* is defined in Gamma *et al.*, **Design Patterns.**

a more general design review is needed here. Perhaps *DHInputModule* needs a way to signal to an output module that action is needed?

The function `createContentCatalog` writes to `std::cout` on every call. This should at least be turned into writing to the error log with "informational" status, and probably be removed. A properly functioning program needed tell the user every time an action is completed successfully.

A.2 *DHOutputModule*

The header for the class *DHOutputModule* forward-declares the class *DHInputModule*, and the source file for *DHOutputModule* includes the header for *DHInputModule*, but this physical coupling seems to be unnecessary; *DHOutputModule* never uses *DHInputModule*. It may suffice to remove the forward-reference and inclusion of the header, to destroy the unwanted coupling.

DHOutputModule is considerably simpler than *DHInputModule*, and has fewer implementation problems. The greatest ones are similar to some of those in *DHInputModule*: poor error handling (*e.g.* printing to `std::cerr` rather than using the error logger facility, and returning `AppResult::OK` even after failures), and poor use of inheritance, resulting in the use of many dynamic_casts in the code.

A.3 *DHOutputStream*

Class *DHOutputStream* is another complex class. It contains approximately 30 member data, most of which are of built-in types; this is generally a sign that some significant abstractions have been missed – and thus the design is more difficult to understand, and thus maintain, than it needs to be. In the header, these data members are grouped into what appears to be logical categories. It may be that refactoring the design to replace these groups with classes (with relevant member functions!) would simplify this class considerably. For example, the lengthy initialization list in the constructor of *DHOutputStream* contains many "magic numbers"; these would be much more easily understandable in the context of a smaller group of cohesive classes, each of which has its own sensible default initializer.

As a second example, in the `DHOutputStream::open` function, we find a complex test which would greatly benefit from the refactoring suggested above. This single test directly involves several policies (file sizes, disk space reservation, detailed pathname manipulation); it is thus fragile, needed change (or at least investigation) when any of these policies change. A suitable refactoring would put each of these behaviors into its own class, which would then be the single point of support for modification of those policies.

Finally, *DHOutputStream* contains a nested class, *EmptySections*, which is clearly and *Edm* concept; *EmptySections* should be removed to *Edm*, or if there is already an equivalent concept there, that class should be used.

A.4 *EventInfoBase* and *EventInfoEntry*

The reason for the existence of the class *EventInfoBase* (in the data handling package) is unclear; it seems that this class contains EDM information, and should be a part of the *Edm* package; if a suitable replacement for it in *Edm* already exists, it should be removed and the *Edm* class used in its place.

If these two classes are to be moved to the package *Edm*, a few modifications are in order. *EventInfoBase* is used as a base class, but is unsuitably designed. It contains no virtual functions, which is a clear sign that inheritance from it is probably misplaced. The destructor for *EventInfoBase* is not virtual, and thus polymorphic use of this class will likely lead to memory corruption or leaks, if not now, in the future. Finally, I could find only one class that inherits from *EventInfoBase*, *EventInfoEntry*. This seems to be a needless complication. If only one derived class is to exist, the complexity of an inheritance hierarchy should be reconsidered; this would remove the problems with the inappropriateness of *EventInfoBase* as a base class.

The implementation of *EventInfoBase* is full of switch statements, based upon the "record type" of the object. In contrast with the needless inheritance introduced with *EventInfoBase*, it seems that this was a place where needed inheritance was omitted. The record within the *EventInfoBase* object would sensibly be replaced by a pointer to a base class, which would be filled in with an instance of one of the subclasses.

The function `printRecordType` does not do what the name implies; it prints nothing, but instead returns a C-style string. Misleading function names like this makes reading, and thus maintaining, the code more difficult. The function name should be changed to reflect its behavior. Additionally, returning a bare pointer (to a null-terminated byte array) imposes a burden on the user, who needs to understand the semantics of the returned pointer. Does it confer ownership of the pointed-to memory, or does it not? It would be simpler to return an instance of the class *std::string*. If efficiency is a concern, then the function should contain one (or more) function-level static strings (so that they are constructed only once), and the return type can be changed to be a const reference to the contained string.

Also in this class, `operator<` is rather complicated; in this function, we have not only switch statements, but *nested* switch statements. Such a function is extraordinarily difficult to understand, and thus very difficult to get right. Furthermore, it is extraordinarily difficult to

test.

This operator takes on unusual importance for this class because *EventInfoEntry* (the sole derived class) is used in a *set* in another class. It is far from clear whether this function induces a strict weak ordering; if it does not, then the *set* into which the *EventInfoEntry* instances are inserted will misbehave in a manner that will be difficult to identify and diagnose. Furthermore, the fact that this operator is called frequently (by the set) makes the inefficiency of the several function calls involved in evaluating the operator especially painful.

A.5 *FileContentCatalog*

The FileContentCatalog.hh presents several classes: *EventInfoEntry*, *LastEntry*, and *FileContentCatalog*. *EventInfoEntry* is described in the section above.

FileContentCatalog contains a *set* of *EventInfoEntry* objects, as well as a pointer to a nested class, *LastEntry*. *LastEntry* contains a pointer to a set of *EventInfoEntry* objects, as well as an iterator into a *set* of *EventInfoEntry* objects. This is an extremely difficult relationship to understand, and is thus a drawback to maintainability. In short time I was able to study this class, I could not clearly discern the purpose for this intertwining of classes. Both are in need of adequate documentation; with such, it may be easier to determine how to simplify this part of the design.

FileContentCatalog seems to function as a collection of "Event" and "BOR" objects. It has several members which perform the function of returning pointers to these objects. Again, the organization is difficult to follow, but it seems again to be inefficient. Returning either and "Event" or a "BOR" requires construction of a temporary *EventInfoBase* object, and a complicated search which includes searching through a the set of *EventInfoEntry* objects to find one that matches the just-constructed temporary. Again the complexity of the code will make testing and maintenance difficult, and again this is exacerbated by the lack of documentation.

As an example of what this complexity leads to, both the member functions `getEvent` and `getBOR` are very inefficient. Each begins with making a temporary *EventInfoBase* object, which is then passed to a function (`getEntry`), where it is used to create a temporary *EventInfoEntry* object, which is then used in a somewhat complicated nest of tests. Refactoring of the design is needed again here.

A.6 Testing

One of the important features required of *DHMods*, stressed by Fedor in the review meetings (see § 2, item 21), is the robustness of the modules.

Specifically, it is important that the crash/recovery code is thoroughly tested. I was not able to find the test code that demonstrates that this testing has been done. If the tests exist elsewhere, I believe they should be moved into the *DHMods* package. If they do not exist, they must be written.

Because of the complexity of the design, and the need for refactorization, it may not be possible to implement such tests immediately. I would urge that writing the tests be a part of the redesign process.

Another functionality noted as important is the "include" and "exclude" support (see § 2, item 2). I was not able to find tests in place that assure these rules (which are rather complicated) are behaving as expected. If the tests exist elsewhere, they should be moved into the *DHMods* package. If they do not exist, they must be written.